Parallelized VLSI Design Rule Checking using CUDA

By Augustin Cavalier, March 2014

Abstract

Currently, most design rule checks (DRC) for VLSI circuits are run on CPUs with very few threads. The purpose of this project was to implement open-source software that performs bitmap-based DRC on both the CPU and on a parallel GPU and to measure the speedup.

What appeared to be one project turned out to be two. I expected to easily locate an open-source toolkit capable of rendering chip design files to use in this DRC project. However, the only open-source chip readers were tied into VLSI design tools. In order to complete the project, I had to write software to render chip designs. This proved to be a project-within-a-project, and required a significant portion of the effort of the planned project. I wrote a checker in CUDA C that partitions the work across CUDA blocks and threads, and avoids warp divergence.

Benchmarks were run using a MIPS CPU design as a test case. In benchmarking my design rule checker, I found that the checker took 27 seconds on the CPU and 2.6 seconds on the GPU, a tenfold speedup.

This project constitutes two major achievements: a library capable of rendering CIF files, and a confirmation that bitmap-based design rule checking is much faster on the GPU.

Table of Contents

I. Purpose

Design rule checking (DRC) is the process by which very-large-scale integrated (VLSI) silicon chip designs are checked before manufacturing. The rules are set by the foundry contracted to make the chip. Due to the fact that chip designs have millions or even billions of transistors, this process can take hours[1]. Polygon-based DRC on a massively parallel GPU (graphics processing unit) has been tried[2], and bitmap-based methods have been discussed before[3,4,5], but no implementations that run bitmap-based DRC on a GPU were found.

I expected the bitmap-based method to be much faster than polygon-based DRC and also much faster than single-threaded DRC on the CPU (central processing unit). The reason for speedup is twofold: bitmap-based DRC is done via a linear algorithm rather than checking polygons, and massively parallel processing is inherently faster than single-threaded CPU processing.

The purpose of this project was to implement open-source software that performs bitmap-based DRC on both the CPU and on a parallel GPU and to measure the speedup.

II. Background: Overview of Design Rule Checking

Method 1: Polygon-based

The polygon-based method of running design rule checks consists of iterating through all the objects in a layer and running math operations on the coordinates. This method has been around for quite some time and is used in a number of free VLSI design software packages[6,7]. The advantage of this method is its accuracy, but the disadvantage is that it is slower than the bitmap-based method. It is nearly always used as the method for the final check of a chip before it is manufactured.

Method 2: Bitmap-based

A second method of doing design rule checks, not as developed or as widely used as the polygon method, is the bitmap method. At least two proposed bitmap-based methods are in existence: a multistage pipeline that uses techniques called "eroding" and "dilating" to remove all except the erroneous parts[8] and a second method that uses "bit-masking" to detect the parts that are too close or too far[9]. Neither of these appears to have become a commercial product.

In the simplest version of this method, a two-dimensional array of pixels ("bitmap") representing one layer of the chip is rendered from the individual polygons, and then checks are run by counting individual pixels and ensuring the counts are at or above the minimum.

III. Background: Overview of Massively Parallel Processing

A technology that has recently come to widespread availability is "massive parallelization", which means running thousands of threads at once on a symmetric multiprocessor ("SMP"). The most common form of SMPs today is in Graphics Processing Units ("GPUs"), which are in desktop PCs.

1 George Taylor, John Ousterhout, "Magic's Incremental Design-Rule Checker". Page 1.
2 Jeremy Espenshade, Michael Romero. "CUDA Independent Study Final Paper".
3 Robert Lougheed, Trevor Mudge. "Design rule checking using serial neighborhood processors".
4 Larry Seiler, "A Hardware Assisted Design Rule Check Architecture".
5 R. Alan Eustace, Amar Mukhopadhyay, "Deterministic Finite Automaton Approach to Design Rule Checking for VLSI".
6 John Ousterhout et al, "Magic Tutorial #6: Design-Rule Checking".
7 Jürgen Thies, "LayoutEditor's Design Rule Checker".
8 Lougheed, Mudge.
9 Seiler.

Modern GPUs utilize one or more symmetric multiprocessors that are capable of running one assembly instruction with a large number of different pieces of data at once, a technique called "same instruction, multiple data" or "SIMD". As long as no branching code is used (such as "`if`", "`switch`", or "`goto`"), SIMD processors outperform CPUs. The NVIDIA company has created an API ("Application Programming Interface") called CUDA ("Compute Unified Device Architecture")[10] that allows their GPUs to be used as a general-purpose multiprocessor.

There is a competing API called OpenCL (Open Computing Language) created by Apple Computer and now managed by the Khronos consortium[11]. OpenCL works on both NVIDIA and AMD Corporation GPUs, but it was not familiar to me and a bit harder to learn, so I selected CUDA instead.

IV. A Method for Running Pixel-based Design Rule Checks using CUDA

A. Introduction

A prototype implementation of a polygon-based design rule checker has previously been implemented and shown to be much faster than its CPU equivalent[12]. Since the polygon-based method does not work as well with SIMD architectures (due to having to schedule individual objects or regions and different object types), I opted to implement a bitmap-based method.

Initially, I expected to easily locate an open-source toolkit capable of rendering chip design files to use in this DRC project. However, the only open-source chip readers were tied into VLSI design tools. In in order to complete the project, I had to write software to render chip designs. This proved to be a project-within-a-project, and required a significant portion of the effort of this project.

B. Materials

Besides the chip rendering library, there were a few other materials that needed to be collected before a DRC algorithm could be written: a chip design suitable for testing, and a viewer capable of examining the chip. Additionally, the computer used for development and testing contains a 64-bit AMD Phenom II processor and a NVIDIA GeForce 550 GPU.

The chip design I selected was a 32-bit MIPS CPU that follows the MOSIS foundry rules and was designed by students at Harvey Mudd College in 2007. It was freely accessible on the Internet, with full documentation and layout files[13]. The layout files were in a text-based format called Caltech Intermediate ("CIF"), using scalable lambda-based design methodology. I chose this format to implement in the chip reader library that I wrote.

I found three freely available tools that could load CIF files: Magic, a free layout editor with roots in the 80s[14]; LayoutEditor, a piece of trialware[15]; and Electric, the system the selected chip design was made on and the most substantial of the three[16]. I used all three in various degrees throughout the project for viewing the chip file.

V. Creating a Chip Reader Library

The Wikipedia article for Caltech Intermediate Format documented the format enough to build a library without any other information. I named the library "ChipLib". It is made up of four

10 NVIDIA Corporation, "CUDA Zone: About CUDA".
11 Khronos Group, "OpenCL".
12 Jeremy Espenshade, Michael Romero.
13 Harvey Mudd College, "E158 Spring 2007 MIPS Project".
14 Tim Edwards, "Magic VLSI Layout Tool".
15 Juspertor UG, "The LayoutEditor".
16 Static Free Software, "About Electric".

components: a parser, an interpreter, a graphics primitive manager, and a renderer. It consists of nearly **1,000 C++ statements**, all heavily debugged, optimized, and tested.

A. Structure

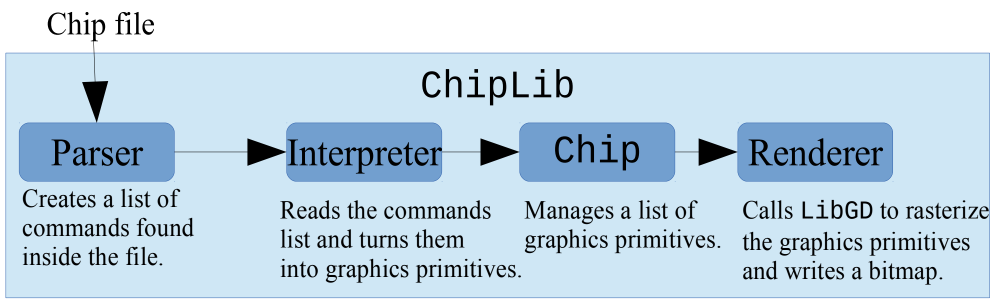

Chip file
ChipLib
Parser
Creates a list of commands found inside the file.
Interpreter
Reads the commands list and turns them into graphics primitives.
Chip
Manages a list of graphics primitives.
Renderer
Calls LibGD to rasterize the graphics primitives and writes a bitmap.

Figure 1: ChipLib's structure

The parser iterates through the file, and builds a list of all the commands in the file. It uses string and list template classes from the Qt toolkit[17] to easily handle comparisons and store commands. In testing, it was shown to parse the 6MB test file in about 3 seconds.

The interpreter reads the list of commands and executes them, including the subroutine calls, scaling, and rotations. It converts them into a list of graphics primitives that are stored by the "`Chip`" class. Initially, it took 30 seconds to load the test chip file, but after optimizing the code to use a lookup table for common angles of sine and cosine, it went down to 13 seconds.

The renderer iterates through all the graphics primitives and rasterizes them using "`LibGD`", a library designed for drawing simple two-dimensional graphics[18]. The renderer writes out the bitmap in the 1-byte-per-pixel format used by the design rule checker. It takes 3 seconds to render the CMF layer of the test file at 27675 by 27675 pixels (2 pixels per λ), and 25 seconds to write out the 730MB file.

It is theoretically possible to modify the renderer to use the GPU for rendering, and to leave the frame-buffer on the GPU for the design rule checker to use. However, I did not have enough time to learn the necessary APIs to do this.

17 The Qt Project, "Homepage".
18 Pierre Joye, "GD Graphics Library".

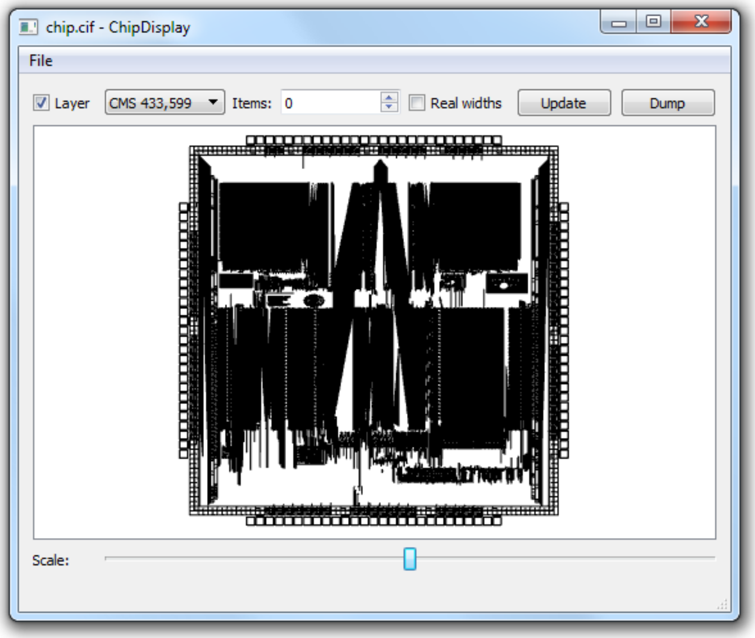


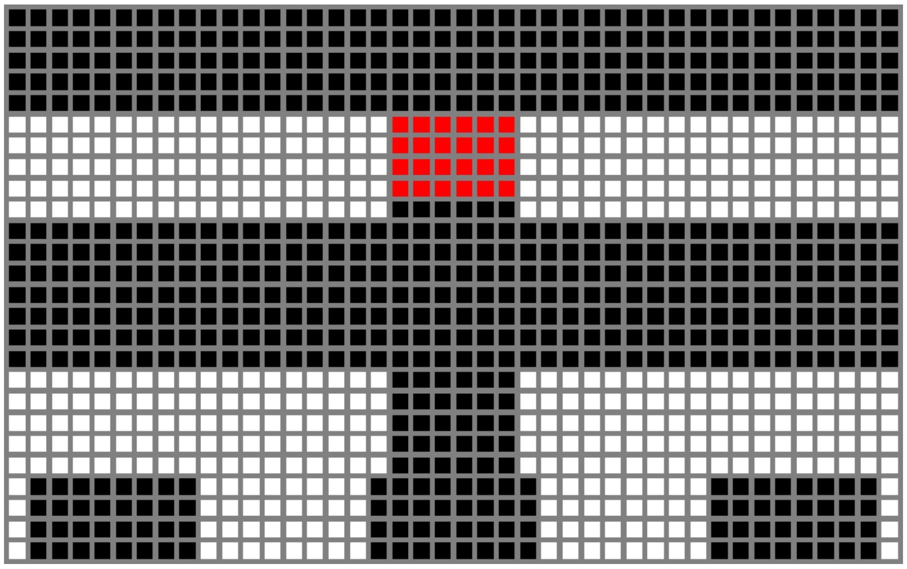
Figure 2: Screenshot of the viewer

I also wrote a graphical user interface for ChipLib that can view and export rendered CIF files. It supports layer filtering, item limits, and dumping the view into vector or bitmap formats. It is also used for creating the rendered bitmap that DRC is executed on.

B. Fixing a fencepost error

While examining the rendered file as part of other testing, a serious fencepost programming error when drawing rectangles was discovered: all rectangles were 1 pixel too tall and 1 pixel too wide. This doesn't sound serious, but when the rectangles are only supposed to be 6 pixels by 6 pixels, this is a significant error. The CIF specification defines rectangles as an X, Y, width, and height. Initially, I had ChipLib transform these rectangles into four-point closed polygons. Upon investigation, I found out that LibGD draws every pixel of the points of the polygon: which is one pixel wider and higher than the CIF rectangle. In order to fix this problem, I changed ChipLib to store the rectangles instead of converting them to polygons, and ensured the renderer would not draw the extra pixel.

C. Error margins



Spacing for this layer should be 5px or greater.

🟥 = Violating cell ☐ = Acceptable cell ■ = Circuitry trace

Figure 3: Example error in the chip file.

The renderer benefits from the MOSIS foundry's "½ lambda grid rule", which states that all SCMOS chip layers must snap to a grid equal to half the lambda of the chip[19]. According to the MIPS chip's documentation, the chip has a physical size of 4mm by 4mm or 13 kλ by 13 kλ[20]. A simple calculation reveals that λ = 0.3 microns, which is 30 CIF units in the test file.

The chip was rendered at a scale of 15 CIF units (0.15 microns) to 1 pixel (1λ = 2 pixels) so that the metalization layer ("CMF", which uses 3λ spacing rules) has a minimum spacing of 6 pixels. Since the chip is rendered at $^1/_{15}$ scale, all spaces greater than 15 CIF units will be rendered as spaces. Because of the ½ lambda grid rule, there should be no spaces smaller than 15 CIF units in this file. Spaces smaller than 15 CIF units are rendered as connections which the checker cannot detect as errors.

VI. Implementation of the Design Rule Checker

A. Introduction

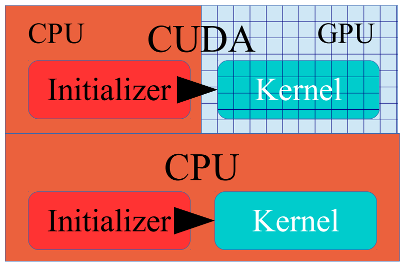


Figure 4: The checker's architecture

There are two components of the design rule checker: the initializer and the kernel, both are implemented in C and support two modes of design rule checking: CPU mode and CUDA mode. The initializer runs on the CPU in both modes, but the kernel runs on the GPU in CUDA mode. The operation of the two components in the two modes is different, as explained in the table. In total it

19 MOSIS, "MOSIS Scalable CMOS Revision 8".
20 Harvey Mudd College, "E158 Spring 2007 MIPS Project – Chip Report".

consists of nearly **600 C statements**.

	CPU mode	CUDA mode
Initializer	Allocates memory.	Allocates memory, copies the bitmap to the GPU, and copies error reports back.
Kernel	Uses a single thread to iterate through the bitmap and create the list of errors.	Uses up to 12,288 threads to iterate through the bitmap and create a list of errors.

Table 1. The design rule checker's modes.

B. Checking Algorithm

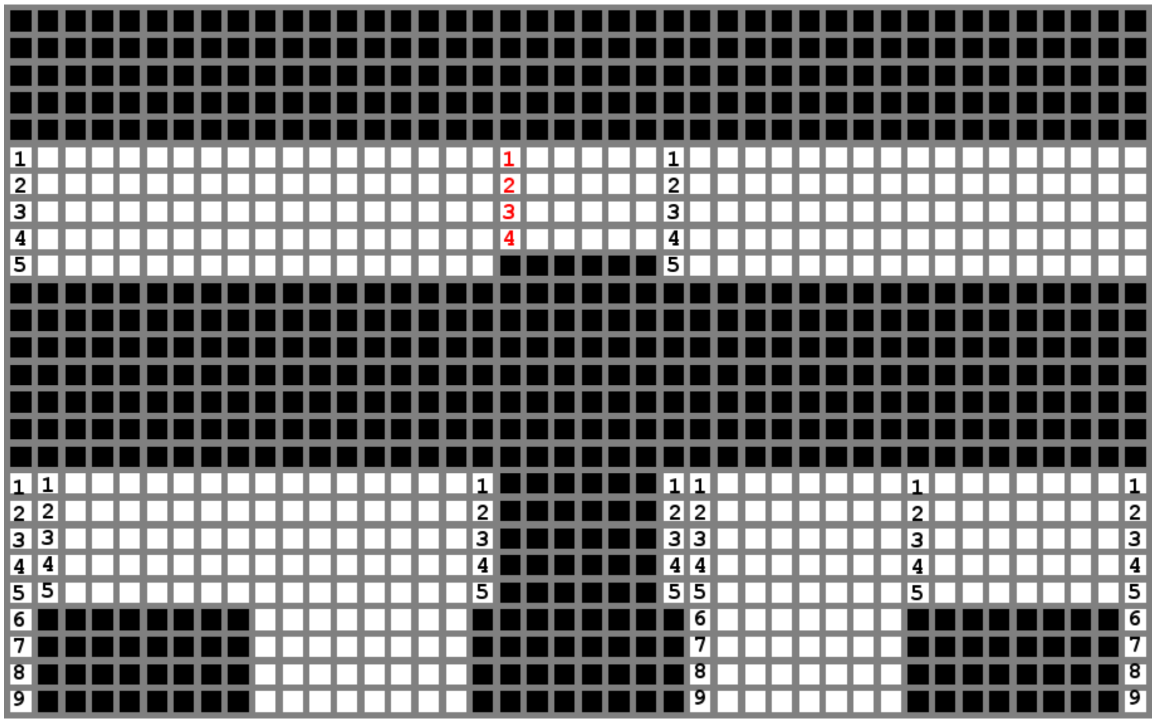


Figure 5: The checking algorithm

The row/column-checking algorithm implements a single check: minimum distance between features of the chip. Spacing rules apply to many of the layers in VLSI designs, but the metal-1 and metal-2 layers of the test chip were used for test purposes.

The heart of the kernel is a "`for()`" loop that goes through each pixel in the row (or column) and identifies which pixels are black ("`1`") and which are white ("`0`"). It keeps track of how many pixels have passed since the last black pixel, and if the spacing is 4 pixels or less it inserts a record into the error list stating the type of violation (vertical or horizontal) and the X and Y coordinates. This record is later presented to the user.

C. Avoiding warp divergence

"Warp divergence" is when one or more threads or blocks in a SIMD group (in CUDA, a "warp") hit conditional code (like the "`if`", "`switch`", or "`goto`" operations mentioned above) and not all the threads choose the same path. When this occurs, the GPU must pause some threads as others run a different path. This takes more time than if the execution path was uniform.

Writing algorithms to avoid possible warp divergence is difficult. Invariant-path mathematical and boolean operations must be used instead of branching code for "`if`" tests, short-circuit operators, and ternary operators. The CUDA compiler and CUDA instruction set have some tricks of their own.

Consider the following C-like pseudocode from an early version of my kernel, which was the code that ran for each and every pixel to detect minimum spacing violations:

```
if(currentPixel == BLACK_PIXEL) {
   if(pixelsSinceBlack < MINIMUM_SPACING && pixelsSinceBlack != 0) {
      errorBuffer[errorBufferIndex] = HORIZONTAL_SPACING_IS_TOO_SMALL;
      errorBuffer[errorBufferIndex+1] = xCoord;
      errorBuffer[errorBufferIndex+2] = yCoord;
      errorBufferIndex += 3;
   }
   pixelsSinceBlack = 0;
} else {
   pixelsSinceBlack++;
}
```

There are two "if" tests which create branching paths. A second source is the use of the "&&" operator. In C and C-like languages, the "&&" operator is short-circuited – if the first condition evaluates to "`false`", it skips the second test and jumps to the "`else`" clause immediately. As written, there are actually three potential branches in this code.

The CUDA compiler is able to prefer branch invariant code generation, but it needs some hints in order to generate branch-invariant code for this example. In some cases, CUDA's "`set.cc`" group of operations can support a ternary ("`?:`") operator without branching. For example:

```
int isFilled = (currentPixel == BLACK_PIXEL) ? 1 : 0;
```

The code above will set "`isFilled`" to "`1`" if the current pixel is black. The ternary operator is shorthand for an "`if`" test in the code. However, since the operator is setting a variable and does not call any functions, CUDA actually has a "conditional set" assembly instruction that it uses so that this code does not become a branch. I utilized this in the later version of my kernel.

The last problem in the code is that a violation must be reported only when there is one. So that this does not cause warp divergence, I eliminated the divergence by running "`reportViolation`" on every iteration, but *not incrementing the write location until an error was found*. So that this would not falsely report an error, I also had to add code at the completion of the loop that cleared the error at the last index position.

Below is the equivalent pseudocode after I reworked it:

```
int isFilled = (currentPixel == BLACK_PIXEL) ? 1 : 0;
int increment = ((pixelsSinceFilled < MINIMUM_SPACING) &&
                 (pixelsSinceFilled != 0)) ? 3 : 0;
increment = isFilled*increment;
errorBuffer[errorBufferIndex] = HORIZONTAL_SPACING_IS_TOO_SMALL;
errorBuffer[errorBufferIndex+1] = xCoord;
errorBuffer[errorBufferIndex+2] = yCoord;
errorBufferIndex += increment;
pixelsSinceFilled = (pixelsSinceFilled+1)*(1-isFilled);
```

When this is compiled with optimizations set to level 2 or higher, the CUDA assembly ("PTX") has no branches within the body of the loop. The CPU assembly ("x86_64"), compiled with GCC set to optimization level 2, still has branches within the body of the loop.

D. Scheduling Algorithm

CUDA kernels work in two-dimensional grids of "blocks" and "threads", where each thread is an instance of the checker, and a block is a group of threads which all issue instructions simultaneously. The number of blocks and threads per block is set by the user. (It is possible to programmatically determine the optimal number of blocks and threads per block, but this is not currently implemented.) Internally, CUDA fires instructions in warps of 32 blocks, so the number of blocks should always be a multiple of 32. If the number of blocks is *not* a multiple of 32, the remaining blocks will be unused computing power.

When the GPU checker is launched, each thread is automatically assigned to a block and given an ID. Block and Thread IDs are two-dimensional coordinates, so in order to assign specific rows and columns to each thread, a thread ID that is unique across all blocks is computed.

Each thread works on a unique set of rows and columns determined by the following information:

(1) its unique thread ID, assigned sequentially (0-N) where N is the total number of threads
(2) the total number of threads
(3) the image's dimensions.

It begins by checking the row equal to its thread ID. It then skips rows by the total number of threads, and continues until there are no more rows in the image. It then waits for all the other threads to complete their horizontal checks, and then all the threads proceed to vertical checks and schedule themselves using the same method.

E. Results

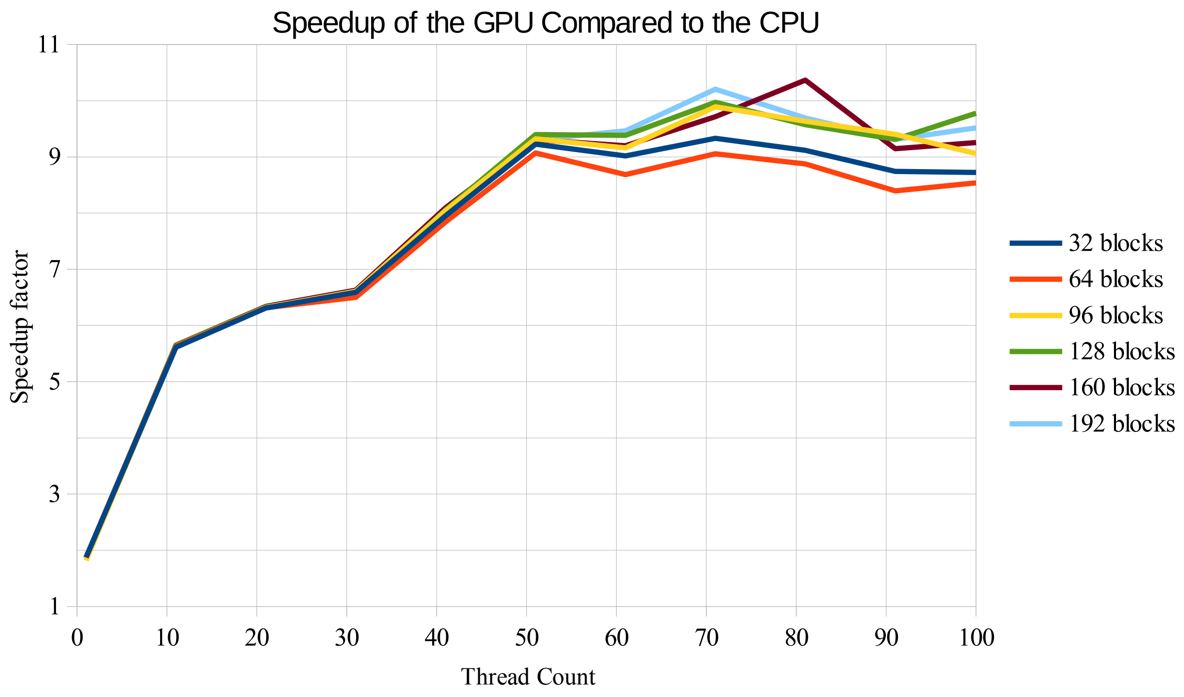


Figure 6: Speedup of the GPU Compared to the CPU

The MIPS CPU's "CMF" layer at 1/15 scale (27675 by 27675 pixels) was the file used for testing. The final benchmarks conducted on March 11th used block sizes of 32, 64, 96, 128, 160, and 192, each with thread sizes of 11, 21, 31, 41, 51, 61, 71, 81, 91, and 101. Each combination was run 10 times and the average was used in the graph. The benchmarks were run on a GeForce GTX 550 GPU on a desktop PC running Linux and the official NVIDIA drivers version 319.

The peak point in the chart above (which indicates the greatest speedup) occurs at 160 blocks and

81 threads. The CUDA checker takes 2639 ms with this configuration. The CPU version of the checker 27353 ms – so the CUDA version is about 10 times faster than the CPU version.

I would have liked to compare my results against a commercial design rule checker, but I was not able to gain access to one for testing.

VII. Conclusion

I had set out to create a design rule checker, but a key component that I thought would be easily available was not. In order to continue with the project, I wrote a software library to read CIF files myself. I completed this project-within-a-project and wrote a software library, ChipLib. I intend to publish this library as open-source software at the conclusion of the science fair. It will be a major asset to anyone who needs to render or examine CIF files since it produces pixel-for-pixel accurate rendered output.

The purpose of this project was to implement open-source software that performs parallel pixel-based DRC on both the CPU and on a GPU to make the checking process faster. The design rule checker I created is useful in that it contains a reference implementation and an algorithm that works on the GPU in parallel and on the CPU. I intend to publish it as open-source software after the science fair.

My testing demonstrates that pixel-based parallelized design rule checking can be up to ten times faster than CPU checking. Therefore, the central goal of the project has been achieved.

For the Future

More time would have allowed the investigation of:

- mapping image coordinates to file coordinates (so that errors could be located within the chip design file)
- checking files that are too large to fit in GPU memory space
- reducing memory usage by using a 1bpp image format instead of 8bpp.
- having the renderer provide a warning when spacing will be rendered as connections
- programmatically determining the optimal block and thread count
- implementing more rules
- supporting the use of multiple GPUs at once.

Acknowledgments

I want to thank:

- My parents, Nina & Forrest Cavalier, for their help and support
- Ed Harris, Ph.D., for the initial project idea, answers for some of my questions, and helpful discussions.
- Jeff Trull, for reviewing the paper and a helpful discussion.

Bibliography

George Taylor, John Ousterhout. "Magic's Incremental Design-Rule Checker". Published 1984. Accessed March 13, 2014. http://cs.york.ac.uk/rts/docs/DAC-1964-2006/PAPERS/1984/DAC84_160.PDF

Jeremy Espenshade, Michael Romero. "CUDA Independent Study Final Paper". Published December 2008. Accessed November 19, 2013. http://halogenica.net/wp-content/uploads/2010/12/CUDA_DRC_Paper.pdf

Robert Lougheed, Trevor Mudge. “Design rule checking using serial neighborhood processors”. Published January 1982. Accessed November 19, 2013. http://google.com/patents/US4441207

Larry Seiler. “A Hardware Assisted Design Rule Check Architecture”. Published 1982. Accessed November 19, 2013. http://cs.york.ac.uk/rts/docs/DAC-1964-2006/PAPERS/1982/DAC82_232.PDF

R. Alan Eustace, Amar Mukhopadhyay. “Deterministic Finite Automaton Approach to Design Rule Checking for VLSI”. Published 1982. Accessed November 19, 2013. http://cs.york.ac.uk/rts/docs/DAC-1964-2006/PAPERS/1982/DAC82_712.PDF

John Ousterhout et al. “Magic Tutorial #6: Design-Rule Checking”. Published February 3, 2008. Accessed November 19, 2013. http://opencircuitdesign.com/magic/tutorials/tut6.html

Jürgen Thies. “LayoutEditor's Design Rule Checker”. Published December 8, 2012. Accessed November 19, 2013. http://layouteditor.net/wiki/DesignRuleChecker

NVIDIA Corporation. “CUDA Zone: About CUDA”. Accessed March 13, 2014. https://developer.nvidia.com/about-cuda

Khronos Group, “OpenCL: The open standard for parallel programming of heterogeneous systems.” Published 2014. Accessed March 22, 2014. http://khronos.org/opencl/

Harvey Mudd College. “E158 Spring 2007 MIPS Project”. Published 2007. The original went offline sometime recently, but the Internet Archive has a copy. Accessed April 30, 2012. http://web.archive.org/web/20120430184224/http://www4.hmc.edu:8001/Engineering/158/07/project/index.html

Tim Edwards. “Magic VLSI Layout Tool”. Published May 7, 2013. Accessed March 13, 2014. http://opencircuitdesign.com/magic/

Juspertor UG. “The LayoutEditor”. Published 2013. Accessed March 13, 2014. http://layouteditor.net/

Static Free Software. “About Electric”. Accessed March 13, 2014. http://staticfreesoft.com/electric.html

The Qt Project. “Homepage”. Published 2013. Accessed March 13, 2014. http://qt-project.org/

Pierre Joye. “GD Graphics Library”. Published May 24, 2013. Accessed March 13, 2014. http://libgd.bitbucket.org/

Metal Oxide Semiconductor Implementation Service (MOSIS). “MOSIS Scalable CMOS Revision 8”. Published May 11, 2009. Accessed March 13, 2014. http://mosis.com/files/scmos/scmos.pdf

Al Danial, “CLOC – Count Lines Of Code”. Published August 16, 2013. Accessed March 13, 2014. http://cloc.sourceforge.net/